



PROGRAMMING Expert

How can you draw in the .NET framework when it has no graphics commands? Easy, says Mike James – it's all a question of classes and objects

PROGRAMMING NEWS

VISUAL STUDIO 2005 MISSING ALM

Application lifecycle management (ALM) may be the big new idea in programming, but Visual Studio developers will have to wait until early 2006 to get it.

ALM helps programmers manage software projects and finished applications, and Microsoft's ALM, Team Studio, was supposed to ship with Visual Studio 2005. However, Team Studio is now expected to be only a beta version in the November launch.

Early versions of Team Studio had lots of bugs, which have presumably delayed its completion. Perhaps Microsoft needs an ALM product to help it manage its ALM project. See www.microsoft.com for more information.

PROGRAM FOR GOOGLE

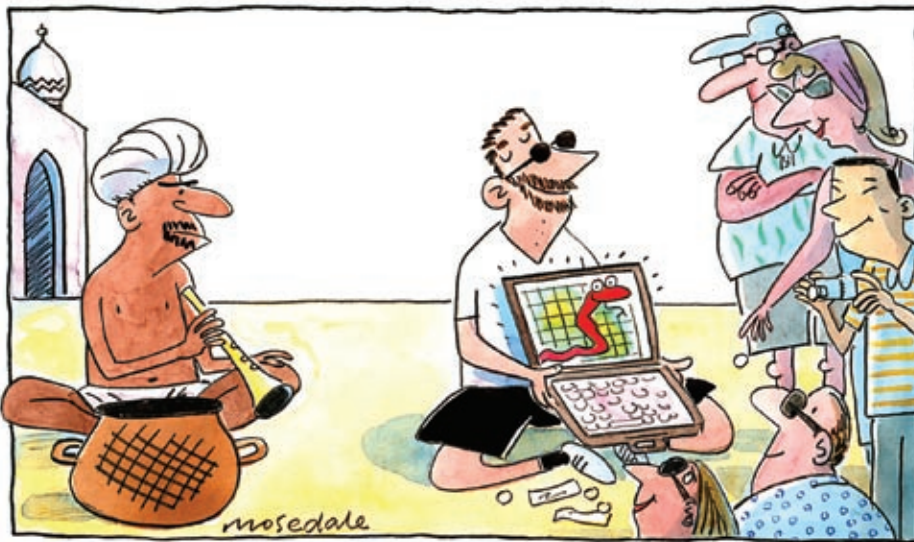
Google has published a slew of application programming interfaces (APIs) to help programmers produce software based on its services. Each API enables you to build Google technology into your applications.

Currently there are APIs for the Google Desktop, Desktop search, Gmail, Froogle, Groups, Earth, Maps, News, Web Search and Google's new instant messaging program, Google Talk. This is similar to Microsoft's Messenger but is based on an open standard called Jabber, which makes it possible to create your own Google Talk client from scratch. See <http://code.google.com> for details.

SKYPE'S THE LIMIT

Skype, the well-known internet phone and instant messaging company, has released details to help programmers work with its communications application. The API is published on the web and you can start using it as soon as you have read the documentation and made sense of it all.

Possible uses include adding internet telephony to an existing application and extending its instant messaging capabilities. See http://share.skype.com/developer_zone for details.



One of the questions I'm asked most frequently by beginners, and even experienced programmers who switch to .NET, is where are the graphics commands? VB 6 and many other languages have specific commands for drawing circles and so on, but you won't find any mention of graphics commands in C# and VB .NET, no matter how hard you look. This is because .NET takes a completely different approach to graphics that enables you to use the same facilities from any language.

The .NET Framework has a number of graphics classes that do the job graphics commands used to do. What's more, the new classes provide a more complete graphics environment than the old VB 6-style commands. The only minor problem is that the entire approach is object-oriented. This is good in the long run but it can make it hard to get started.

So with this in mind, let's look at how easy it is to work with graphics in .NET. We'll start by looking at some basic techniques and then apply them to a simple game, Snake. Along the way we'll cover such topics as using the GDI+ to draw 2D graphics; the differences between a class and an object; the use of the keyword 'this'; how to make graphics persistent; and using a Bitmap as a buffer to produce flicker-free animation. For our game we'll also use keyboard events to get real-time user input.

GDI+

If you have programmed low-level graphics before, you may know about Windows' Graphic Device Interface (GDI). This low-level API gives programs access to the fundamental graphics facilities in Windows and the .NET Framework classes. You can

regard GDI+ as an object-oriented extension of this. If you don't know anything about the GDI, don't worry, because the GDI+ is easier to use and more powerful.

The examples are all written in C# using the C# Express Beta 2, which you can download for free from Microsoft's website (www.microsoft.com). As VB .NET gives you access to exactly the same GDI+ objects, it is easy to convert the examples to VB .NET or any other .NET language.

As the GDI+ is object-oriented, everything you use is either an object, a method or a property. Graphics objects include such things as pens, while their properties deal with aspects such as colour, thickness and so on.

The most important GDI+ object is embodied in the Graphics class. This is a drawing surface and provides a host of methods and properties for creating graphics. Any object created from the Graphics class must be associated with a physical graphics device (such as a screen) or a portion of it (such as a form) when the object is first created.

All Windows objects on which you can draw generally have a graphics object associated with them. This makes things seem a little more complicated at first, but it soon makes sense. For example, a common requirement is to draw on a form, and to do this you simply access the form's graphics object.

YOUR FIRST DRAWING

Start a new Windows application and place a button on the form. To show clearly how the graphics object works, we'll create another form just to draw on. In the button's click event handler, enter the following:



```
Form Form2 = new Form();
Form2.Visible = true;
Form2.Activate();
```

The first line creates a new form object, Form2, from the Form class. The next two lines make it visible and activate it, so it has the focus and is the topmost window.

Next we need to create a graphics object that is connected to the form's drawing surface:

```
Graphics G = Form2.CreateGraphics();
```

To make this work, add the following line to the other 'using' statements:

```
using System.Drawing.Drawing2D;
```

Once you have a graphics object, you can start to draw using a collection of other objects and methods that will quickly seem self-explanatory. For example, the pen and brush objects set the outline and fill properties for the graphics object. Many of the drawing methods accept a pen or brush as parameters. For example, the DrawLine method is:

```
DrawLine(Pen, Point, Point);
```

where Pen is an instance of a pen object and Point is a point object. If you want to draw a red line between two points, you would use:

```
Pen p = new Pen(Color.Red);
Point start = new Point(10, 10);
Point end = new Point(100, 100);
G.DrawLine(p, start, end);
```

Creating objects that you use only once – such as p, start and end – is a bit of a nuisance. The easy alternative is to create them dynamically as you need them, although this results in very long method calls. For example, you can write the previous example as:

```
G.DrawLine(new Pen(Color.Red),
    new Point(10, 10),
    new Point(100, 100));
```

This does the same job without you having to invent names for variables, and the objects are disposed of as soon as the method is complete.

Once you understand pens and points, you can start looking at the documentation for the graphics objects and properties you can use. For example, a pen can have a width as well as a colour:

```
p.Width = 5;
```

There is also a collection of predefined pen objects to save you the trouble of creating one. For example, to specify a standard black pen, just use:

```
Pens.Black
```

The SystemPens class returns a relevant pen object for the style used to draw particular Windows components. For example, SystemPens.Control gives the pen used to draw the outline colour of a button or control.

What about shapes other than straight lines? The graphics object has additional methods for

GRAPHICS OBJECT METHODS FOR DRAWING SHAPES

OUTLINE SHAPES	FILLED SHAPES
DrawArc	
DrawBezier(s)	
DrawClosedCurve	FillClosedCurve
DrawCurve	
DrawEllipse	FillEllipse
DrawLine(s)	
DrawPie	
DrawPolygon	FillPolygon
DrawRectangle(s)	FillRectangle(s)

both solid and outline shapes, as you can see in the table above.

Two more useful methods are Clear, which clears the drawing area to a specified colour, and FillRegion, which fills the interior of a region.

All these methods work as you'd expect and are well documented. You may also encounter some additional geometric objects for defining the location and size of some of the shapes. For example, a bounding rectangle object specifies the size of shape you want to draw when using methods such as DrawEllipse. You can specify the rectangle in a number of ways, but the simplest is to provide the coordinates of its top left-hand corner plus its height and width. For example:

```
G.DrawEllipse(Pens.Black, new Rectangle
    (10, 20, 100, 200));
```

A QUESTION OF CLASS

If you try the techniques shown above to draw on the form containing your button, you will encounter a problem that often puzzles beginners but is easy to solve. You may think that Form1 is just like Form2 and write:

```
Graphics G = Form1.CreateGraphics();
```

This won't work because CreateGraphics creates a graphics object for an object, but Form1 is not an object, it's a class. Classes provide all the instructions for making an object, but the object isn't created until you instantiate the object using the 'new' keyword or, in the case of the main form class, by running your application.

When you write a program using any of the Visual .NET languages, the main code you write is generally a class. Typically, it's the class for the main form that appears when your application starts and has the default name Form1. The Form1 class inherits everything it needs to be a form from the .NET framework Form class, and you then add additional methods and properties. The designer does this for you when you add buttons and other controls. In short, you customise the default blank form in order to create a new class.

The designer automatically generates the code that creates a form object from the class. If you look inside the program code, you will see that the last line is:

```
Application.Run(new Form1());
```

This creates a new Form1 object without giving the new object a name. As your class definition can't know the name that an object based on the class will have, you can't refer to the object by name. Instead you use the keyword 'this'. Within a class

'this' means 'the current object instance created using the class'. So in the Form1 class, instead of:

```
Graphics G = MyObjectName.CreateGraphics();
```

you use:

```
Graphics G = this.CreateGraphics();
```

In Visual Basic the Me command does the same thing. In both languages, these keywords clearly demonstrate the difference between a class and an instance of the class, and it is worth thinking about the distinction until you are completely sure you understand what is going on. When you're ready, try changing the example program to draw on the form that has the button on it.

PAINTING BY NUMBERS

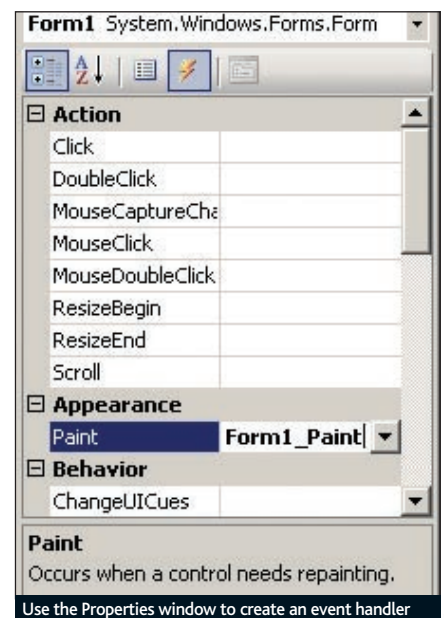
When you try any of the examples, you will notice that if you minimise your form and then restore it, your drawing disappears. In other words, it isn't persistent. This is because Windows' drawing surfaces are not saved automatically. You draw on them and then something else can draw over them.

If you want to restore what you drew, you have to draw it again. The system tells you when redrawing is necessary – for instance, when a window on top of your form closes. This is what the Paint event and its handler are all about. The Paint event is triggered when the form needs redrawing, and you put any graphics that need to be redrawn in the event handler.

The easiest way to create an event handler for a C# form is to go to the designer and select the Events tab in the Properties window. Double-click on the event in question to generate the basic code needed for an event handler. In this case you need to select the form class, Form1, and the Paint event. This generates an event-handling function with the correct parameters:

```
private void Form1_Paint(object sender, PaintEventArgs e)
```

When the Paint event is triggered, the handler is passed the PaintEventArgs object, called e, which





REVIEW BOOK

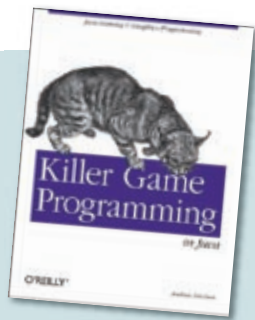
Killer Game Programming in Java

RATING ★★★★★

PRICE £32

ISBN 0596007302

PAGES 996



Killer game programming in Java? You must be joking. Java is slow, cumbersome and quite unsuitable for writing graphics and sound in real time. However, Andrew Davidson does an excellent job of trying to convince us otherwise. There are no Java compilers for the PlayStation or Xbox, but there are plenty of PCs out there, and many new mobile phones and PDAs use Java. There are real earning opportunities for Java programmers who know the graphics and sound techniques used in games.

This book is very specific to Java, and few of its insights can be generalised to anything else. It explains a few general principles, but most of it is about working with Java's 2D, 3D and sound APIs to create 2D and 3D graphics. The examples aren't realistic, but then you wouldn't want them to be – the book is long enough as it is. Program examples are well described but they often lack an overview that explains the object of the exercise.

If you want to know how to create graphics and sound, this is an excellent book. It doesn't try to teach you about Java or object-oriented programming and gets started with its main subject matter almost at once.

✓ A good introduction to graphics and sound in Java

✗ The ideas don't generalise well

REVIEW BOOK

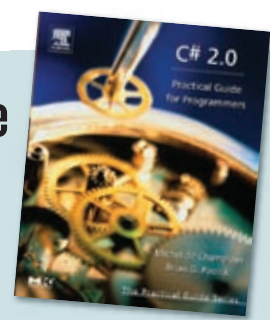
C# 2.0: Practical Guide for Programmers

RATING ★

PRICE £14

ISBN 0121674517

PAGES 272



At first glance this book looks promising. It has a nice cover and is well laid out. Sadly, this impression doesn't last long and doubts start to surface as soon as you look at the detail. The examples are too long and the explanations of some of the syntax use Backus Naur Form, which is fine if you are an expert but doesn't help otherwise. It's also surprising that such a short book should have two authors, Michel de Champlain and Brian G Patrick. The overall result is an uninspiring account of the superficial features of C# 1.0 with a few comments about using C# 2.0 thrown in.

Overall it reads like a reference work and provides very few insights into how features work and how to make the most of them. Rather than any sort of sympathetic explanation, it simply presents the facts and assumes the reader will understand. It certainly doesn't deal with the Framework of classes that causes 99 per cent of the problems when working in .NET, and nor does it deal with Visual Studio or C# Express. In fact, the only thing to say in its favour is that it's short and inexpensive. Other than that, this book has nothing to recommend it.

✓ Slim, cheap reference on C#

✗ Tedious, technical and too focused on version 1

includes a graphics object that you can draw on. To draw a persistent line and an ellipse, you would write something like this:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics G = e.Graphics;
    G.DrawLine(Pens.Black,
        new Point(10, 10), new Point(100, 100));
    G.DrawEllipse(Pens.Black,
        new Rectangle(10, 20, 100, 200));
}
```

PERSISTENCE MADE EASIER

Using the Paint event to make graphics persistent is the textbook way of doing things, but it's not the best. In many cases you simply want to draw whenever it suits you and have the result preserved when the form is hidden and then redisplayed. It isn't always easy to put all your drawing commands in the Paint event handler.

The usual solution is to do your drawing on a bitmap held in memory and use the Paint event handler to copy the bitmap to the form (a process called a BitBlt).

This is a useful technique, but in practice there is an even easier way. Every form has a BackgroundImage property, which you can set to a background image that is displayed automatically whenever the form is repainted. This is easy to use, but you must be careful if it's to work properly. First, create a bitmap object of the correct size:

```
Bitmap BImage=new Bitmap(this.ClientSize.Width,
    this.ClientSize.Height);
```

This creates a bitmap object that's the same size as the client area of the form.

Next we need to make this new, empty bitmap the background image for the form:

```
this.BackgroundImage = BImage;
```

From this instruction on, the bitmap becomes the form's background. To draw on it, we need to associate a Graphics object with it. We construct this using the static Graphics object:

```
Graphics GBack = Graphics.FromImage(BImage);
```

Now that we have a Graphic object, we can use all its drawing commands as before and leave the Form object to render the bitmap whenever it needs repainting. For example:

```
GBack.DrawEllipse(Pens.Black, new Rectangle(0, 0, 100, 200));
```

Now anything you draw to the GBack graphics object is persistent.

You don't have to create the additional objects BImage and GBack. The code below makes and disposes of these objects on the fly to achieve the same result, although it's a bit harder to follow:

```
this.BackgroundImage = new Bitmap(this.ClientSize.Width,
    this.ClientSize.Height);
Graphics.FromImage(this.BackgroundImage).DrawEllipse(
    Pens.Black,
    new Rectangle(0, 0, 100, 200));
```

You have to set the form's BackgroundImage

property to a Bitmap object only once. After this you can get a Graphics object as often as you like and use it to draw on the form.

MAKE A SNAKE

The old computer game Snake underwent a deserved revival a few years ago, mainly because it works well on portable devices such as mobile phones. It is not a hard game to program and is a good way to learn more about the GDI and graphics in general.

Start a new C# Windows project called Snake and place a timer control on it. We will divide the screen into a grid of blocks representing possible locations for the snake's head or a body segment. In this case, each block is 10x10 and the screen is 50x50 blocks. To make this information available to the rest of the program we need some global constants, variables and objects:

```
public partial class Form1 : Form
{
    const int cx=10,cy=10;
    Size CSize = new Size(cx, cy);
    Size SSize=new Size(50*cx, 50*cy);
    int vx, vy;
```



It may not look much, but the red curve runs around the screen in a snake-like way



We are using Size objects to store the width and height of a shape. The variables vx and vy are used to store the snake's horizontal and vertical velocity. The snake is made up of little boxes and the location and size of each is stored in an array of Rectangle objects:

```
Rectangle[] Snake = new Rectangle
[50];
int Head = 15, Tail = 0;
```

The snake's head is initially stored in Snake[Head] and the tail in Snake[Tail]. We also need a global Image object (to hold a bitmap) and a global Graphics object:

```
Image OffScreen;
Graphics GForm;
```

We can use the form's constructor to initialise everything:

```
public Form1()
{
    InitializeComponent();
}
```

First we set the form's client area to the correct size for the game:

```
this.ClientSize = new Size(SSize.Width,
SSize.Height);
```

We could use the background bitmap to make the graphics persistent, but updating this rapidly produces flashing, so we'll use a bitmap buffer to build up the new graphic and transfer it to the form when we are ready. The bitmap must be the same size as the client area of the form, and we also need the graphics object associated with the form so that we can transfer the bitmap to it later:

```
OffScreen = new Bitmap(
    this.ClientSize.Width,
    this.ClientSize.Height);
GForm = this.CreateGraphics();
```

We could initialise all 16 locations of the snake's body separately but in practice we can start simply by setting the head's position and the size of all the other array elements:

```
Snake[Head].Location = new Point
(11 * CSize.Width, 7 * CSize.Height);
for (int i = 0; i <= Snake.
    GetUpperBound(0); i++)
    Snake[i].Size = CSize;
```

Using an array of Rectangle objects, it's easy to set the location with a Point object and the size using Size object.

Finally we initialise the snake's velocity and the timer:

```
vx = 1; vy = 1;
timer1.Interval = 100;
timer1.Enabled = true;
```

Most of the rest of the work goes on in the timer's Tick event handler. Use the Properties dialog box to add this event handler to your program automatically. The first thing to do is calculate the

new position of the snake's head based on its current position by adding the velocity scaled by the size of each graphics block. The only problem is that you don't want the snake to disappear off the edge of the form. The simplest solution is to 'wrap' the form around so that going off one edge brings the snake back on the opposing edge.

We can achieve this using modular arithmetic and the modular arithmetic operator %. With modular arithmetic, if you exceed a number you specify, that number is automatically subtracted from the result to bring it back in the modular range. For example, $5 + 6 \% 8 = 3$, because the result 11 exceeds 8 by 3. In our case the limits are $50 * cx$ in the X axis and $50 * cy$ in the Y axis.

```
private void timer1_Tick(object
    sender, EventArgs e)
{
    Point NewHead = Snake[Head].
        Location;
    NewHead.X = (NewHead.X + vx * cx +
        50 * cx) %
        (50 * cx);
    NewHead.Y = (NewHead.Y + vy * cy +
        50 * cy) %
        (50 * cy);
```

The new head location is stored in the next array element and the Head pointer increases by one to indicate which array element holds the head. The Tail pointer must also be increased.

The only complication is that we must treat the array as if it were circular to stop it running out of space. This is just a matter of using modular arithmetic again, which makes the head and tail pointers go back to 0 after reaching 49:

```
Head = (Head + 1) % 50;
Tail = (Tail + 1) % 50;
```

Now that we have the position of the new head in the array we can store it:

```
Snake[Head].Location = NewHead;
```

Next we simply blank out the tail position by drawing a block in the form's background colour and draw the new head in a colour of our choice. To do this we need a graphics object created from the bitmap:

```
Graphics G = Graphics.FromImage(Off
    Screen);
G.FillRectangle(SystemBrushes.
    Control, Snake[Tail]);
G.FillRectangle(Brushes.Red, Snake
    [Head]);
```

Finally we draw the bitmap on the form:

```
GForm.DrawImage(OffScreen, 0, 0);
```

If you now run the program, a red snake will slowly appear, segment by segment, and then move off in a fixed direction. When it reaches the edge of the screen it appears on the other side. Seeing it in action should illustrate some of the subtler points of how it works. For example, to animate a snake you have only to draw the new position of the head and blank out the previous tail position. When the

snake starts, it grows by one segment per movement until the tail catches up with the head's original position, when it starts to move.

KEY EVENTS

The next step is to control the direction of the snake using the keyboard. This is easy in C#, but the method is completely different to that in VB 6, because it is event-driven. There is certainly no function like Inkey for testing the current state of the keyboard. Each time you press a key you generate a keypress event, and to interact with the keyboard you must write event handlers. The easiest way is to use the designer. Select events in the properties window and double-click on the Keydown event to generate a suitable handler.

```
private void Form1_KeyDown(object
    sender, KeyEventArgs e)
{
}
```

The KeyEventArgs object e has a KeyCode property, which you can use to discover which key has been pressed. The Keys collection also has a set of built-in keycodes against which you can match the key press. The simplest method is to use a switch statement to deal with cases where the right and left arrow keys have been pressed (the equivalent in VB is the Select statement). When the right arrow key is pressed the snake's velocity rotates clockwise and when the left arrow key is pressed it rotates anticlockwise:

```
int nvx, nvy;
switch(e.KeyCode)
{
    case Keys.Left:
        nvx = Math.Sign(vx + vy);
        nvy = Math.Sign(-vx + vy);
        vx = nvx;
        vy = nvy;
        break;
    case Keys.Right:
        nvx = Math.Sign(vx - vy);
        nvy = Math.Sign(vx + vy);
        vx = nvx;
        vy = nvy;
        break;
}
```

The arithmetic 'rotates' the velocity by 45° each time one of the keys is pressed using a well-known technique based on a rotation matrix. You can easily handle more key presses in the same way.

The program currently lets us take charge of a snake and control it as it moves around the screen. The next step would take more space than we have, but it's basically an application of the same ideas. You need to add obstacles for the snake to avoid, rewards for it to eat and so on. You can use the basic graphics techniques explained here to good effect in almost any program. **CS**

CONTACT

MIKE JAMES

Mike has written several books on computing and programming and has been a senior lecturer at Teesside University

mike@computershopper.co.uk